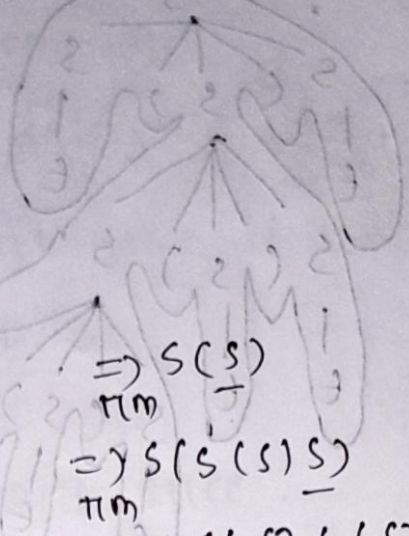
CC (After mid) 1st 03.05.23

construct a parse tree :-

$S \rightarrow S(S)S$

$|\epsilon$

string :- (()())



So,

$S \underset{rm}{\Rightarrow} S(S)\underline{S} \underset{rm}{\Rightarrow} S(S)\epsilon \underset{rm}{\Rightarrow} S(\underline{S})$

$\underset{rm}{\Rightarrow} S(S(S)\underline{S})$

$\underset{rm}{\Rightarrow} S(S(S)S(S)\underline{S})$

$\underset{rm}{\Rightarrow} S(S(S)S(S)\epsilon))$

$\underset{rm}{\Rightarrow} S(S(S)S(\underline{S}))$

$\underset{rm}{\Rightarrow} S(S(S)S(\epsilon))$

$\underset{rm}{\Rightarrow} S(S(S)\underline{S}())$

$\underset{rm}{\Rightarrow} S(S(S)\epsilon())$

$\underset{rm}{\Rightarrow} S(S(\underline{S})())$

$\underset{rm}{\Rightarrow} S(S(\epsilon)())$

$\underset{rm}{\Rightarrow} S(\underline{S}()())$

$\underset{rm}{\Rightarrow} S(\epsilon()())$

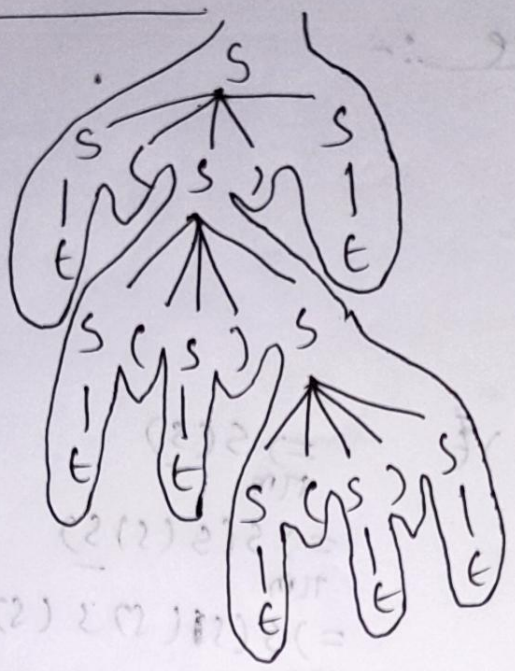
$\underset{rm}{\Rightarrow} \underline{S}(()())$

$\underset{rm}{\Rightarrow} \epsilon(()())$

$\underset{rm}{\Rightarrow} (()())$

P.T.O

So Parse tree :-



~~Str : ε~~

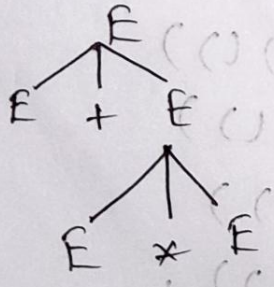
str :- ε (ε (ε) ε (ε) ε) ε

=> (( ) ( ))

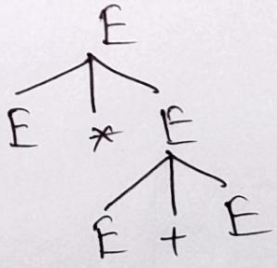
E → E + E
| E * E
| id

id → 2 | 3 | 5

Tree - 1 :-



Tree - 2



* if any grammar generates 2 different parse tree then it is called ambigious grammar.

=) You cannot use an ambigous grammar because it confuses the compiler.

=) compiler need to be deterministic.

A grammar is called ambigous if we find :-

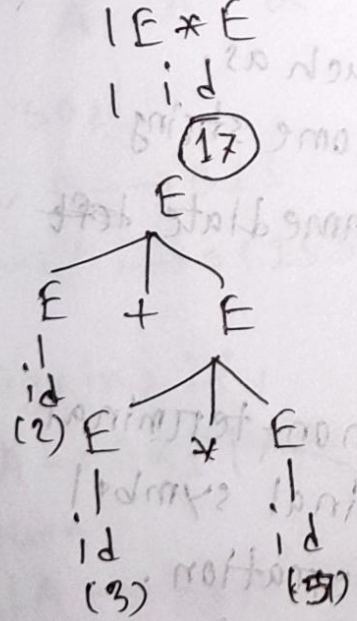
— More than one LMD or
— More than one RMD or
— More than one parse tree.

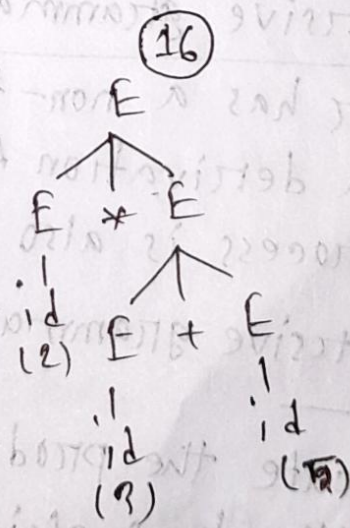
10.05.23

2nd (CC)

Left recursive grammar → Non deterministic grammar.

$E \rightarrow E+E \mid E*E \mid id$





Precedence priyority :- (+, −, *, /, %)

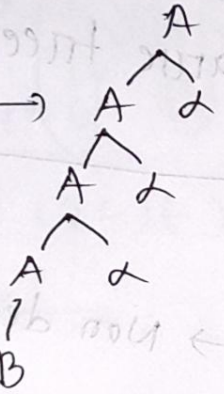
(setting priyority for operators)

* if there is non-terminal after terminal symbol than it is a infinite or left-recursive grammar.

Ex! A() {
    A();
    α;
}

A → Aα | B

P.t. ⟶



R.E :- Bα*

left recursive grammar :-

A grammar has a non-terminal A such as, there is a derivation A ⇒ Aα. for some string α. this process is also known as immediate ~~left~~ left recursive grammar.

Problem :-

(i) Execute the production of a non-terminal symbol without checking the terminal symbol which produce infinite loop or iteration.

(ii) As it executes the production of a non-terminal symbol, so this is the way to

ambiguous.

increase probability to make a grammar ~~ambiguous~~.

(iii) A left recursive grammar makes the iteration with $B\alpha^*$, if the grammer is.

$A \rightarrow A\alpha | B$

we have to eliminate the left recursive grammer. without changing it's iteration $B\alpha^*$. Therefore the grammar should be for.

if, $A \rightarrow A\alpha | B$ is:-

then, equ. imp

$A \rightarrow BA'$

$A' \rightarrow \alpha A' | \epsilon$

$A \rightarrow \alpha A$

Ex:-

$S \rightarrow S0S1S | 01$ (immediate)

$\overline{A} \quad \overline{A} \quad \overline{\alpha} \quad \overline{B}$

so,

$S \rightarrow 01S'$

$S' \rightarrow 0S1SS' | \epsilon$

Ans

Again, if,

$A \rightarrow A\alpha_1 | A\alpha_2 | A\alpha_3 | \ldots | A\alpha_m | B_1 | B_2 | B_3 | \ldots | B_n$

then,

equ imp.

$A \rightarrow B_1A' | B_2A' | B_3A' | \ldots | B_nA'$

$A' \rightarrow \alpha_1A' | \alpha_2A' | \alpha_3A' | \ldots | \alpha_mA' | \epsilon$

Ex:-

$E \rightarrow \underline{E+T} \mid \underline{E*T} \mid \underline{a} \mid \underline{b}$

$\underline{A} \quad \underline{A} \; \alpha_1 \quad \underline{A} \; \alpha_2 \quad \underline{B_1} \quad \underline{B_2}$

then,

$E \rightarrow aE' \mid bE'$

$E' \rightarrow +TE' \mid *TE' \mid \epsilon$

Non-immediate left recursive grammer :-

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \epsilon$

$S \Rightarrow \underline{A}a$

$\Rightarrow \underline{S}da$ .... incomplete.

Question:

1) $L \rightarrow L, S \mid S$

2) $E \rightarrow Tc \mid a$

$T \rightarrow F*T \mid Eyd \mid \epsilon$

$F \rightarrow T \mid id$

3) $A \rightarrow ABd \mid Aa \mid a$

$B \rightarrow Be \mid b$

4) $S \rightarrow A$

$A \rightarrow Ad \mid Ae \mid aB \mid ac$

$B \rightarrow bBc \mid f$

Sol :-

(I) $L \rightarrow L, S \mid S$

$\underline{A} \quad \underline{A}\,\underline{\alpha} \quad \underline{B}$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \epsilon$

(II) $E \rightarrow Tc \mid a$

$T \rightarrow F * T \mid Eyd \mid \epsilon$

$F \rightarrow T \mid id$

1) $E \Rightarrow Tc$

$\Rightarrow Eydc$

$E \rightarrow Eydc \mid a$

$\underline{A} \quad \underline{A}\,\underline{\alpha} \quad \underline{B}$

$\therefore E \rightarrow aE'$

$E' \rightarrow ydcE' \mid \epsilon$

2) $T \rightarrow T * T \mid Tcyd \mid a$

$\underline{A} \quad \underline{A}\,\underline{\alpha_1} \quad \underline{A}\,\underline{\alpha_2} \quad \underline{B}$

$\therefore T \rightarrow aT'$

$T' \rightarrow *TT' \mid cydT' \mid \epsilon$

(III) $A \rightarrow ABd \mid Aa \mid a$

$\underline{A} \quad \underline{A}\,\underline{B}\,\underline{\alpha_1} \quad \underline{A}\,\underline{\alpha_2} \quad \underline{B}$

$B \rightarrow Be \mid b$

$\underline{B} \quad \underline{B}\,\underline{\alpha_1} \quad \underline{B}\,\underline{\alpha_2} \quad \underline{b}$

$\therefore A \rightarrow BA' \mid BeA' \mid bA'$

$A' \rightarrow BdA' \mid aA' \mid a\epsilon$

17.05.23

3nd (CC)

A() {
  A();
  α
} ≈ Bα*

$$A \to BA'$$
$$A' \to \alpha A' \mid \epsilon$$

Ex:-

$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to (E) \mid id$$

Sol:

$$\underset{A}{E} \to \underset{A}{E} \underset{\alpha}{+T} \mid \underset{B}{T}$$

$$E \to TE'$$
$$E' \to +TE' \mid \epsilon$$

again,

$$\underset{A}{T} \to \underset{A}{T} \underset{\alpha}{*F} \mid \underset{B}{F}$$

$$T \to FT'$$
$$T' \to *FT' \mid \epsilon$$

$$X \to Xsb \mid sa \mid b$$
$$S \to Sb \mid Xa \mid a$$
$$X \to Xsb \mid b$$

$$\longrightarrow$$

$$X \to sa \mid b$$
$$X \to Xaa \mid aa \mid b$$

Non-determinism

Non-determinism means you have numerous option on a single symbol. consider the following grammer:-

$$A \rightarrow \alpha B_1 | \alpha B_2 | \alpha B_3$$

From this grammer you have to derive αB_3. In order to derive the grammar you need to perform backtracking. In addition, the backtrack is happened because of common prefixs and it is the concept of non-determinism.

Left Factoring :—

If you have a non-deterministic grammer —

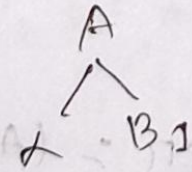
$$A \rightarrow \alpha B_1 | \alpha B_2 | \alpha B_3 | \cdots | \alpha \gamma$$

then the deterministic or the determinism of the grammer could have:-

$$A \rightarrow \alpha A' | \gamma$$
$$A' \rightarrow B_1 | B_2 | B_3$$

$A \rightarrow \alpha A'$

$A' \rightarrow B_1 | B_2 | B_3$



if there is common prefix than it is non-deterministic

Ex:-

$S \rightarrow \underline{bSS}aas | \underline{bSS}asb | \underline{bSb} | a$

Sol/ $S \rightarrow bSS' | a$

$S' \rightarrow \underline{Sa}aes | \underline{Sa}sb | b$

P.T.O →

Again, as again common prefix found.

$S' \rightarrow SaS'' \mid b$

$S'' \rightarrow aS \mid Sb \mid \epsilon$

CC (4th) 18.05.23

Elimination of (left factoring) non-deterministic does not remove the ambiguity :-

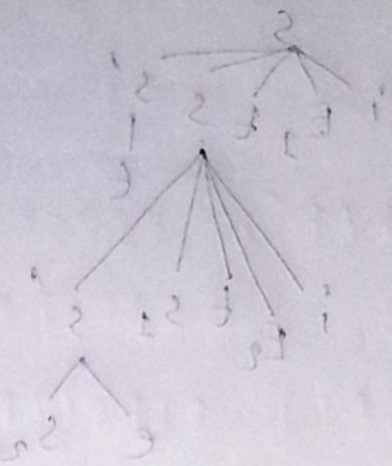
stmt → if expr then stmt
| if expr then stmt else stmt
| a

if → i
expr → E
then → t
stmt → s
else → e

S → iEtS
| iEtSeS
| a

dangling-else grammer :- (After left factoring it also can occur ambiguisness)

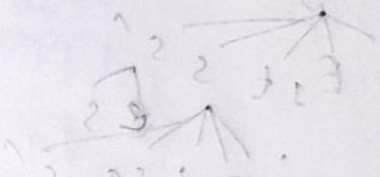
```
if () {
    {
if () {
    {
else {
    }
S → iEts
  | iEtses
  | a
```



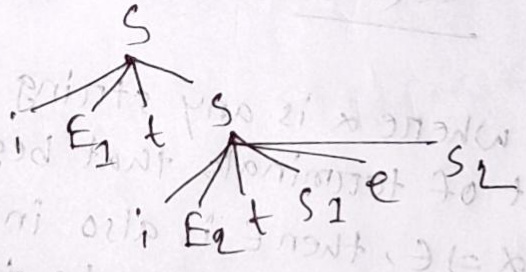
grammer :-

if E_1 then if E_2 then
S_1 else S_2

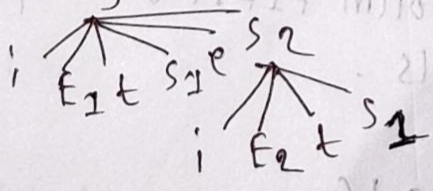
string : $iE_1tiE_2ts_1es_2$



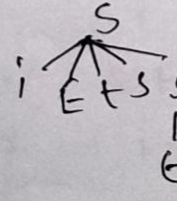
1st production :-

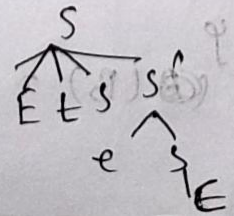


2nd production :-

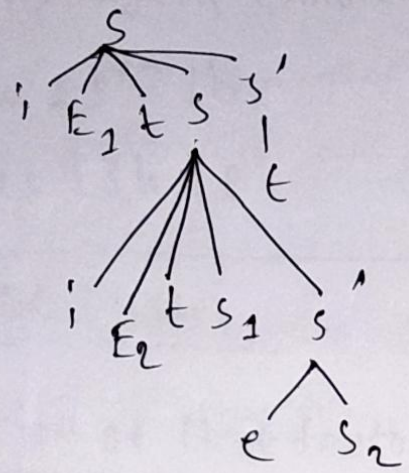


S → iEtss'
s' → ∈ | es | a

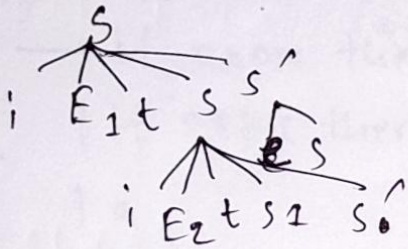




I)



II)



CC (5th) 24.05.23

FIRST :-

FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begain strings derived from α. If $\alpha \Rightarrow \epsilon$, then ϵ is also in FIRST(α).

Nevertheless, you must eliminate the left recursive prior to perform FIRST on any non-terminal or terminal symbols.

I) $A \rightarrow aA$

II) $A \rightarrow a\beta \rightarrow a \mid \epsilon$

III) $A \rightarrow Ba \mid \epsilon$

$B \rightarrow b \mid \epsilon$

FIRST(B) = {b, ϵ}

FIRST(A) = FIRST(B) = {b, ϵ, a}.

first have to check if there is any left recursion or not.

Ex:-

$S \rightarrow ABCD$

$A \rightarrow b \mid \epsilon$

$B \rightarrow D \mid c \mid \epsilon$

$C \rightarrow d$

$D \rightarrow e \mid f \mid \epsilon$

$FIRST(S) = FIRST(A) = \{b\}$

$= FIRST(B) = \{e, f, c\}$

$= FIRST(C) = \{d\} = \{b, e, f, c, d\}$

$FIRST(A) = FIRST(b) = \{b\}$

$= FIRST(\epsilon) = \{\epsilon\}$

$= \{b, \epsilon\}$

$FIRST(B) = FIRST(D) = \{e, f\}$

$= FIRST(c) = \{c\}$

$= FIRST(\epsilon) = \{\epsilon\} = \{e, f, c, \epsilon\}$

$FIRST(C) = FIRST(d) = \{d\}$

$FIRST(D) = FIRST(e) = \{e\}$

$= FIRST(f) = \{f\}$

$= FIRST(\epsilon) = \{\epsilon\}$

Ex:-

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$f \rightarrow id \mid (E)$

6 FIRST(E) = FIRST(T) = {id, (}

FIRST(E') = FIRST(+) = {+}
= FIRST(ϵ) = {ϵ} = {+, ϵ}.

FIRST(T) = FIRST(F) = {id, (}

FIRST(T') = FIRST(*) = {*}
= FIRST(ϵ) = {ϵ}
= {*, ϵ}

FIRST(F') = FIRST(id) = {id}
= FIRST(( ) = {(}
= {id, (}.

Ex:-

S → aBDh
B → cC
C → bC | ϵ
D → EF
E → g | ϵ
F → f | ϵ

FIRST(S) = FIRST(a) = {a}
FIRST(B) = FIRST(c) = {c}
FIRST(C) = FIRST(b) = {b}
= FIRST(ϵ) = {ϵ} = {b, ϵ}

FIRST(D) = FIRST(E) = {g}
= FIRST(F) = {f, ϵ} = {g, f, ϵ}.

FIRST(E) = FIRST(g) = {g}
= FIRST(ϵ) = {ϵ}
= {g, ϵ}

FIRST(F) = FIRST(f) = {f}
= FIRST(ϵ) = {ϵ}
= {f, ϵ}

FOLLOW :-

To compute FOLLOW(A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set:

RULES:-

1) Place $ in FOLLOW(S), where S is the start symbol, and $ is the input right endmaker.
2) If there is a production A → aBβ. then everything in FIRST(B) except ϵ is in FOLLOW(B) [FOLLOW(B) = FIRST(B)]
3) If there is a production A → αB. or a production A → αBβ. where FIRST(B) contains ϵ, then everything in FOLLOW(A) is in Follow(B).

Ex:-

S → ABCD
A → b|ϵ
B → D c|ϵ
C → d
D → e|f|ϵ

FIRST (S) = {b, e, f, c, d}

FIRST (A) = {b, ϵ}

FIRST (B) = {e, f, c, ϵ}

FIRST (C) = {d}

FIRST (D) = {e, f, ϵ}.

→

FOLLOW (S) = {$}

~~FOLLOW (A) = FIRST (A) = {b}~~.

FOLLOW (A) = FIRST (B) = {e, f, c}

= FIRST (C) = {d}

= {e, f, c, d}

FOLLOW (B) = FIRST (C) = {d}

FOLLOW (C) = ~~{e, f}~~ FIRST (D) = {e, f}

~~= FIRST ($) = {$}~~

= FOLLOW (S) = {$} = {e, f, $}

~~FOLLOW (D) = f~~

FOLLOW (D) = FOLLOW (S) = {$}

= FIRST (C) = {c} = {$, c}.

= {$, c}.

Non-terminal symbols	b	e	f	c	d	$
S		S	→	A B C	D	
A	A → b	A	→	ϵ		
B		B →	D c		B → ϵ	
C					C → d	
D		D → e	D → f	~~D → ϵ~~ D → ϵ		D → ϵ

CC (6TH) 31-05-23

LL(1) Grammar :-

The first "L" stands for -
- Scanning the input ~~signal~~ symbol from left to right.

The second "L" stands for -
- Producing left most derivative

The "1" means -
- Using one input symbol of look ahead at each step to make parsing decisions.

For example, consider the grammar -

$S \rightarrow aSb \mid \epsilon$

Now, if we scan each input symbol of this production one by one from left to right (first a, then S and finally b). then the process belongs to the LL(1) concept.

- How to recognise a grammar is LL(1) ?

If we get only one entry on production for each terminal symbol corresponding it's non-terminal symbol on parsing table then that grammar is called a LL(1) grammar.

When a grammar is not LL(1) ?

1. If you get more than one entry for each terminal symbol on parse ~~tree~~ table.
2. If any grammar is left recursive.
3. If any grammar has non-determinism.

Construct a parse ~~tree~~ table for the grammar:-

$E \rightarrow TE'$

$E' \rightarrow +TE' | \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \epsilon$

$F \rightarrow (E) | id$

FIRST(E) = FIRST(T) = {(, id}

FIRST(E') = FIRST(+) = {+}

= FIRST(ε) = {ε}

= {+, ε}

FIRST(T) = FIRST(F) = {(, id}

FIRST(T') = FIRST(*) = {*}

= FIRST(ε) = {ε}

= {*, ε}

FIRST(F) = FIRST(() = {(}

= FIRST(id) = {id}

= {(, id}.

FOLLOW(E) = FIRST($) = {$}

~~= {$ + ε~~ = {$,)}.

FOLLOW(E') = FOLLOW(E) = {$,)}.

FOLLOW(T) = FIRST(E') = {+}.

= FOLLOW(E') = {$,)}.

= {+, $,)}.

FOLLOW (T') = FOLLOW (T) = {+, $,)}.

= FOLLOW (T') = {+, $,)}

= {+, $,)}.

FOLLOW (F) = FIRST (T') = ~~{+, *, ε}~~ {*, ε}.

= FOLLOW (T') = {+, $,)}

= FOLLOW (T') = {+, $,)}.

Non-terminal symbol	(	)	+	*	id	$
E	E → TE'				E → TE'	
E'		E' → ε	E' → +TE'			E' → ε
T	T → FT'	~~[illegible]~~	~~[illegible]~~		T → FT'	~~[illegible]~~
T'		T' → ε	T' → ε	T' → *FT'		T' → ε
F	F → (E)				F → id	

This is an LL(1) grammar.

Ex:-

S → iEtSS' | a

S' → eS | ε

E → b

FIRST (S) = FIRST (i) = {i}.

~~FIRST (S)~~ = FIRST (a) = {a}

= {i, a}.

FIRST(S') = FIRST(e) = {e}

= FIRST(ε) = {ε}

= {e, ε}.

FIRST(E) = FIRST(b) = {b}.

FOLLOW(S) = {$}.

= FIRST(S') = {e}.

= FOLLOW(S) = {$, e}

= FOLLOW(S') = {$, e}.

= {$, e}.

FOLLOW(S') = FOLLOW(S) = {$, e},

FOLLOW(E) = FIRST(t) = {t},

Non-terminal	i	a	e	b	t	$
S	S→iEtSS'	S→a				
S'			S'→eS S'→ε			S'→ε
E				E→b		

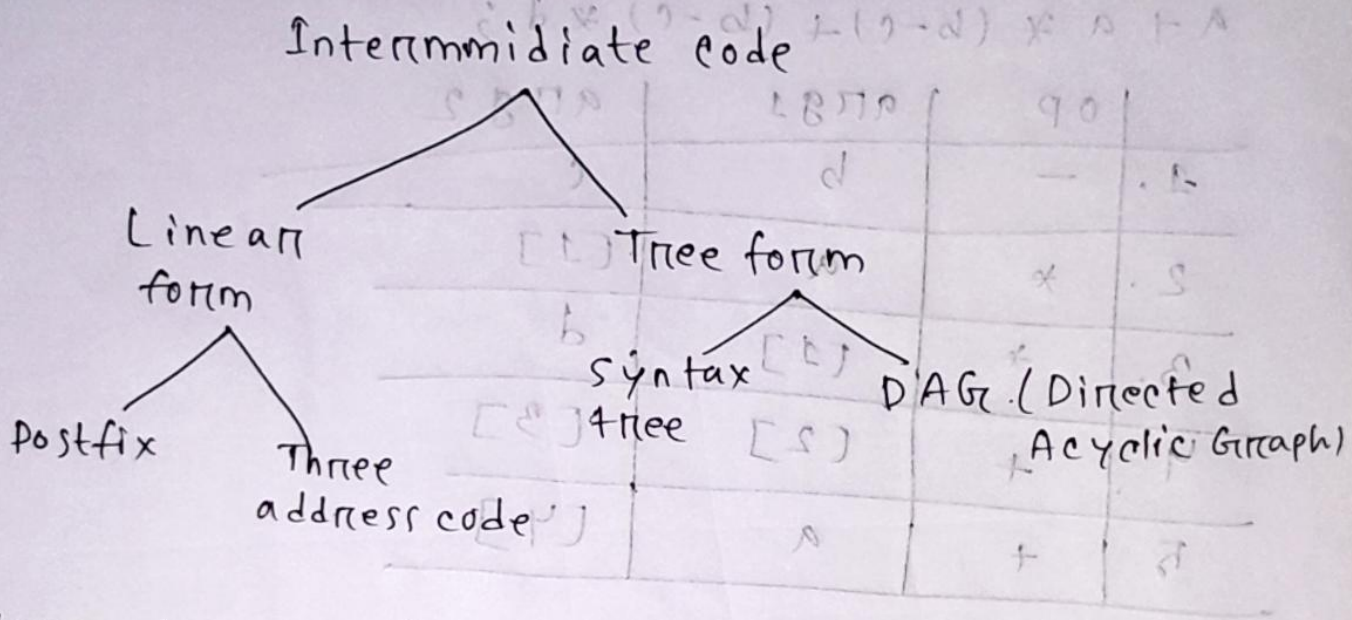
Not LL(1) grammer.

LL(1) → Left to right, left recursive, 1 input

S → aSbS | bSaS | ε [special case]

so better if we find from parse table.

ICG (Intermmidiate code generator)



Three address code :-

Quadruples :- it ~~contains~~ consists of 4 fields, namely : OP, arg1, arg2, and result.

Triples :- it ~~contains~~ consists of 3 fields, namely :- OP, arg1, arg2. [more memory efficiency]

Quadruples :-

Ex :-

a + a * (b - c) + (b - c) * d ;

	OP	arg1	arg2	result
1	-	b	c	t_1
2	*	a	t_1	t_2
3	*	t_1	d	t_3
4	+	t_2	t_3	t_4
5	+	a	t_4	t_5

Triples :-

a + a * (b - c) + (b - c) * d ;

	op	arg1	arg 2
1.	-	b	c
2.	*	a	[1]
3.	*	[1]	d
4	+	[2]	[3]
5	+	a	[4]